Learning to Walk: Reinforcement Learning on FrozenLake
MS&E 338 Winter 2016/2017

SUNet ID: hannahli, tengz
Name: Hannah Li, Teng Zhang

1 Introduction

In this project we empirically compare the performance of several reinforcement learning algorithms. The goal is to provide some insight of the efficiency of different algorithms, and the impact of parameters on their performance.

The project is based on the OpenAI gym toolkit, from which we use the FrozenLake-v0 game environment. Here we give a brief introduction.

1.1 OpenAI Gym

OpenAI Gym is a toolkit for developing and comparing reinforcement learning algorithms on games such as Pacman, Roulette, and Asteroids. Each game is formulated as an episodic MDP with underlying transition and reward matrices that are unknown to the agent.

1.2 FrozenLake-v0

The agent walks on a frozen lake attempting to retrieve a lost frisbee ('G'), the location of which is fixed in the problem setting but is unknown to the agent. Along the way, there are holes ('H') that the agent may fall into, ending the game. The agent must learn where the frisbee is and how to walk to the frisbee. However, the lake is slippery and transitions are stochastic with respect to a state-action pair.

The game ends when agent finds the frisbee or falls into a hole. There is a reward of 1 for finding the frisbee and a reward of 0 otherwise.

2 Algorithms

We implement three algorithms on the problem of FrozenLake, one model-based learning algorithm and two value function learning algorithms.

2.1 PSRL

PSRL is a model-based learning algorithm which, through interactions with the environment, attempts to learn the game by learning the distribution of the transitions $\mathcal{P}_{s,a}$ and the rewards $\mathcal{R}_{s,a}$ for every state-action pair. A reward and transition matrix are sampled from the posterior distribution of the system, optimal policies are calculated for the sampled MDP, and then the results of the policy are used to update the posterior.

In the implementation of PSRL on FrozenLake, for each state action pair the transition prior is Dirichlet$(2/N, \ldots, 2/N)$, where N is the number of states. The reward prior, which will be discussed in the next section, is originally Dirichlet(1,1) in the case where rewards are either 0 or 1 and Dirichlet$(1, 1, 1)$ in the case where rewards are 0, 1, and -k.

Algorithm 1 PSRL

Input: $\mathcal{S}, \mathcal{A}, N, \rho$
for $\ell = 1, 2, 3, \ldots$ **do**
  for all (s, a) **do**
    Compute posterior distribution over $\mathcal{P}_{s,a}$, $\mathcal{R}_{s,a}$
    Sample $\mathcal{P}_{s,a}$, $\mathcal{R}_{s,a}$ from posterior distribution
  end for
  Compute the optimal policy π_ℓ for MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}_{s,a}, \mathcal{R}_{s,a}, \rho)$
  $s_0^\ell \sim \rho()$
  for $t = 0, \ldots, N-1$ **do**
    Apply policy π_ℓ
    Observe $r_{t+1}^\ell, s_{t+1}^\ell$
  end for
end for

2.2 LSVI

LSVI is a value function learning algorithm inspired by the value iteration algorithm for a given MDP realization. Given the buffer, which is the history dataset, LSVI estimates a value function Q_{θ_H}, where Q is a function family parameterized by θ and H denotes the planning horizon. The algorithm iterates through the planning horizon and at each time step h, fits a value function $Q_{\theta_{h+1}}$ using the estimated value function Q_{θ_h} and immediate rewards in the next round.

Algorithm 2 LSVI

Input: Q: value function family parametrized by θ, buffer, λ, v: regularization parameter, H: planning horizon, $\hat{\theta}$: prior mean.
Output: $\tilde{\theta}$
Initialize $\tilde{\theta}_0$ to be null.
for h in $(0, \ldots, H-1)$ **do**
  Regress

$$\tilde{\theta}_{h+1} = \arg\min_{\theta \in \mathbb{R}^D} \left(\frac{1}{v} \sum_{(s,a,r,s') \in \text{buffer}} \left(r + \max_{a' \mathcal{A}} Q_{\tilde{\theta}_h(s',a')} - Q_{\theta(s,a)} \right)^2 + \frac{1}{\lambda} \|\theta - \hat{\theta}\| \right)$$

end for

In our experiments, after using LSVI to calculate Q_{θ_H} we then apply the ϵ-greedy algorithm, acting randomly with probability ϵ and acting greedy with respect to Q_{θ_H} with probability $1 - \epsilon$.

2.3 LSVI_TD

LSVI_TD is the incremental version of LSVI, it takes the essence of mini-batch gradient descent method.

Instead of starting from scratch and solving a full optimization problem every time we learn, we use the current parameter as the starting point, repeatedly sample some data from the buffer and do a gradient step based on these so-called batches.

The reasoning behind this method is that when the data is not sufficiently large, solving the full optimization problem completely does not guarantee a better approximation of the value function than simply solving the problem "partially". Moreover, by implementing the gradient step we dramatically decrease the computation time needed.

Algorithm 3 LSVI_TD

Input: Q: value function family parametrized by θ, buffer, λ, v: regularization parameter, sample: method of sampling from buffer, $\tilde{\theta}$: previous value function parameter, γ: discount factor, α: learning rate, N: number of mini-batches, $\hat{\theta}$: prior mean.

Output: $\tilde{\theta}$

for n in $(1, \ldots, N)$ **do**

batch $\leftarrow$ sample(buffer)

Define

$$\mathcal{L}(\theta) = \frac{1}{v} \sum_{(s,a,r,s') \in \text{batch}} \left(r + \max_{a' \mathcal{A}} Q_{\tilde{\theta}_h(s',a')} - Q_{\theta(s,a)} \right)^2 + \frac{1}{\lambda} \|\theta - \hat{\theta}\|$$

Apply

$$\tilde{\theta} = \tilde{\theta} - \alpha \nabla_\theta \mathcal{L}(\theta)$$

end for

After performing the above algorithm to estimate Q_{θ_H} we again use an ϵ-greedy approach to decide the policy for the episode.

3 Results

We implement the above algorithms onto the FrozenLake game and run experiments on the parameters and reward values to find an algorithm that learns this game well.

3.1 Augmenting rewards

We augment the reward associated with falling into a hole in the learning algorithm, while keeping the reward 1 for finding the goal and 0 for landing in a space that is neither the goal or a hole. We judge the performance of the algorithm using the original rewards (where we get +1 for getting to the goal and 0 otherwise). Tuning the augmented rewards greatly increases the performance of both PSRL and LSVI.

For PSRL, in the case where rewards are 0 or 1, the reward prior is Dirichlet(1,1) for each state-action pair. When we modify the rewards to be 0, 1, or -k, the reward prior is Dirichlet(1,1,1).

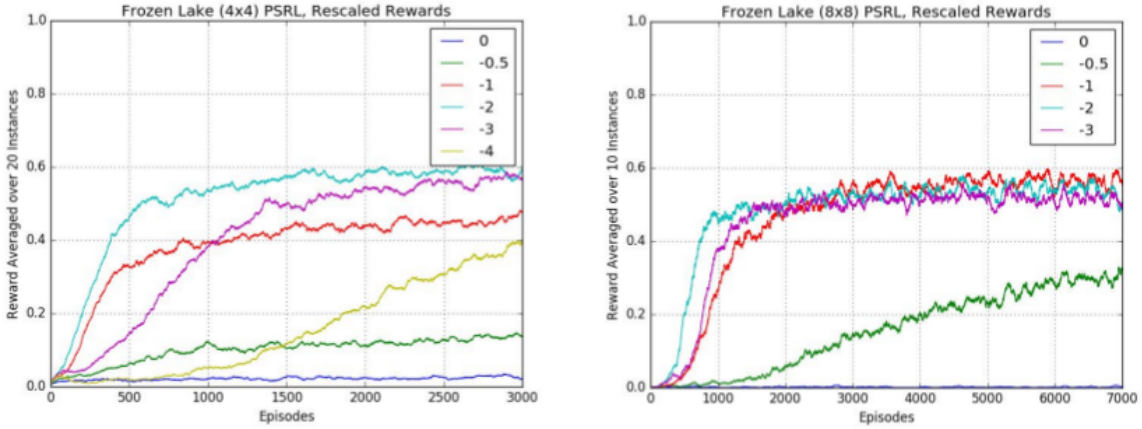

Figure 1: PRSL

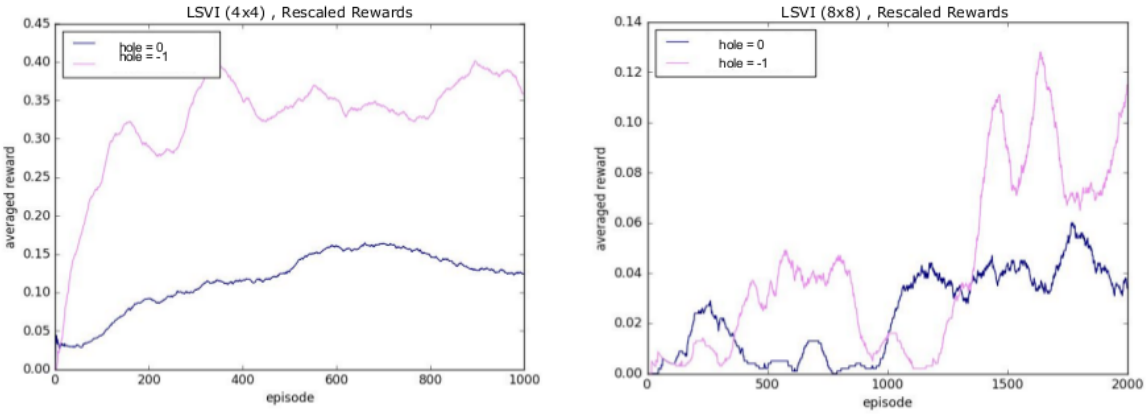

Figure 2: LSVI

3.2 Buffer size and number of batches

We ran experiments on buffer sizes and number of batches for LSVI_TD. The results are shown for a fixed number of batches and varying buffer sizes as well as a fixed buffer size and varying number of batches.

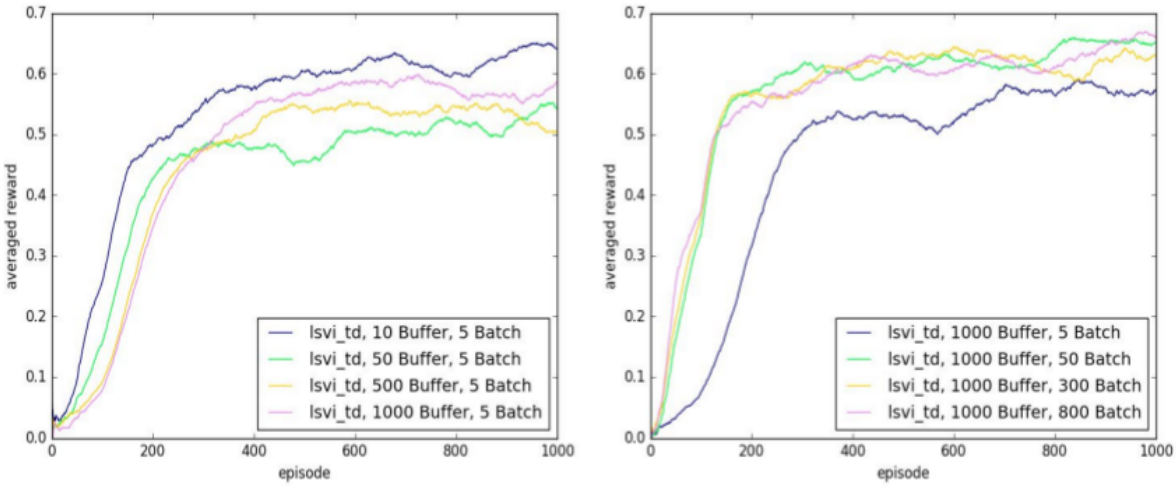

Figure 3: LSVI_TD, 4x4 FrozenLake

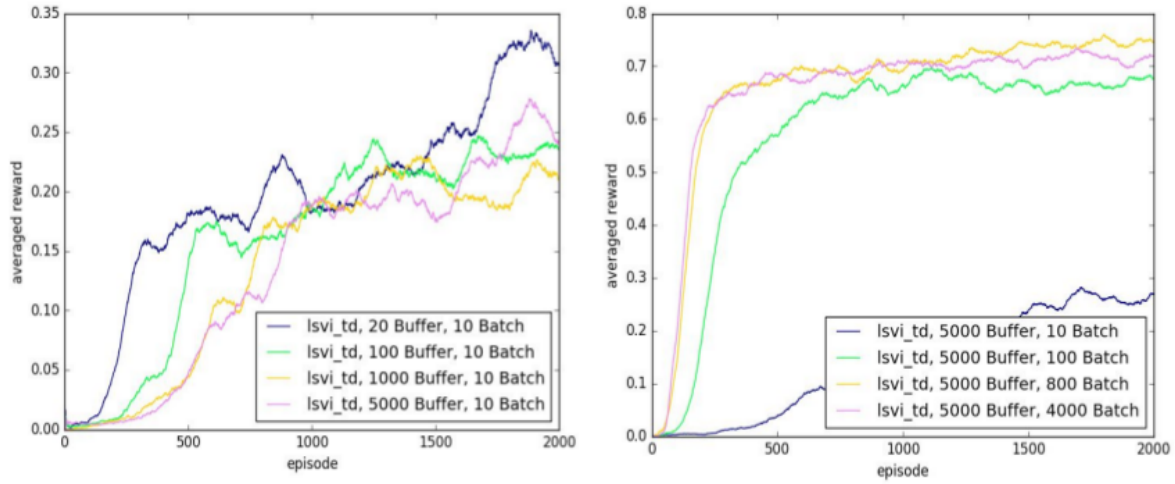

Figure 4: LSVI_TD, 8x8 FrozenLake

We can see that for a fixed number of batches, increasing the buffer size will not help in speeding up the learning process. When the buffer size is fixed, however, increasing the number of batches does help, though the marginal benefit of doing so is decreasing. This indicates that the algorithm essentially needs us to focus more on the most recent data.

3.3 Efficiency of LSVI and LSVI_TD

To reach a level of performance comparative to that of the LSVI_TD algorithm above, LSVI needs a buffer size of at least 5000, while in LSVI_TD, only 100 samples of the 5000 historical data are needed to achieve a similar performance.

Also note that the performance of LSVI is increasing in buffer size, though we see the same diminishing returns that we see in LSVI_TD.

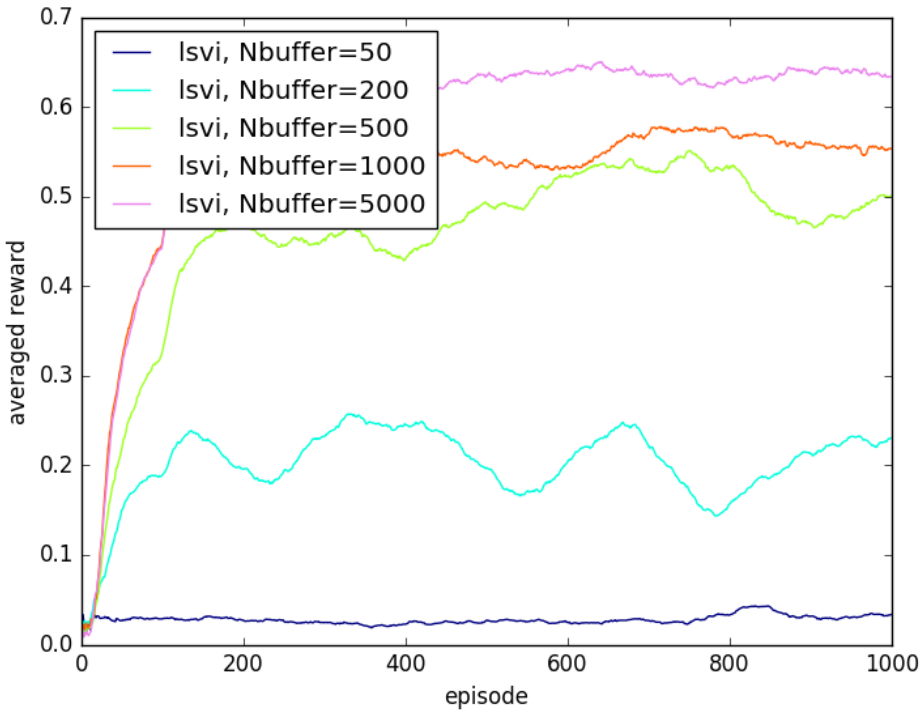

Figure 5: LSVI performance with different buffer sizes

3.4 Other results

We also did experiments on other parameters, including the learning rate α, the discount factor γ of LSVI_TD, and the ϵ in the eps-greedy action function. The results are presented below.

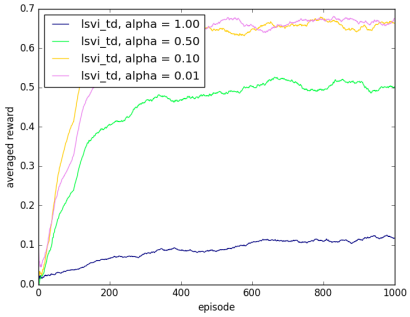

Figure 6: LSVI_TD, α

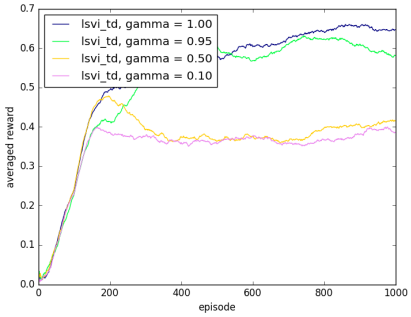

Figure 7: LSVI_TD, γ

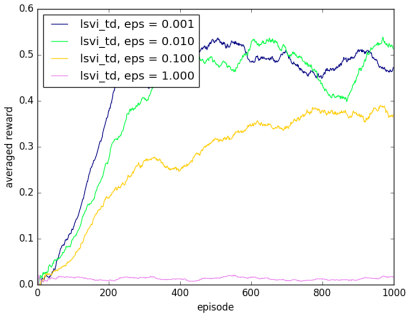

Figure 8: LSVI_TD, ϵ

We can see similar pattern of results that require moderate tuning of the parameters.

4 Discussion

4.1 Efficiency of LSVI and LSVI_TD

In our experiments, LSVI needed a buffer size of at least 5000 to achieve a similar level of performance as LSVI_TD using 100 samples from the buffer size of 5000. LSVI_TD is likely more information efficient for this problem and also has a faster running time.

4.2 Prior belief of the goal reward

In the experiment, we are actually using a misspecified prior, where the agent does not know where the goal is or if the goal will have positive or negative reward. Alternatively, one might assume that the agent knows the location of the goal or that the agent knows that there is a positive reward, in which case our priors are misspecified. To analyze the latter issue, consider a revised version of FrozenLake, where the reward for reaching the goal is -1 or 1 with equal probability.

Under this setting, if the goal reward is 1, the optimal outcome is as above, to reach the goal, and if the goal reward is -1, the optimal outcome is to stay in the system as long as possible. The simulation result shows that LSVI_TD can also learn well under this setting, where the prior we are using is unbiased.

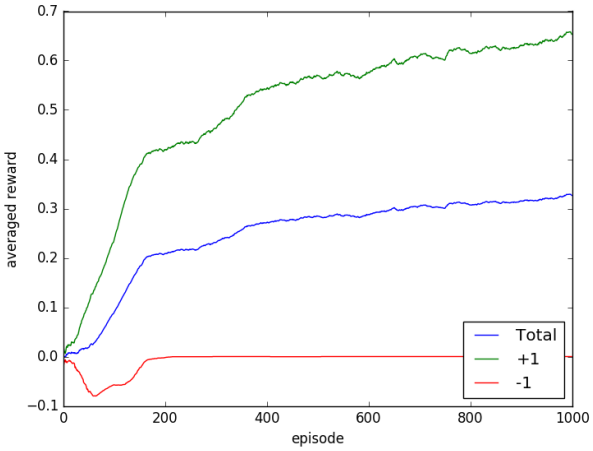

Figure 9: Unbiased prior, average reward

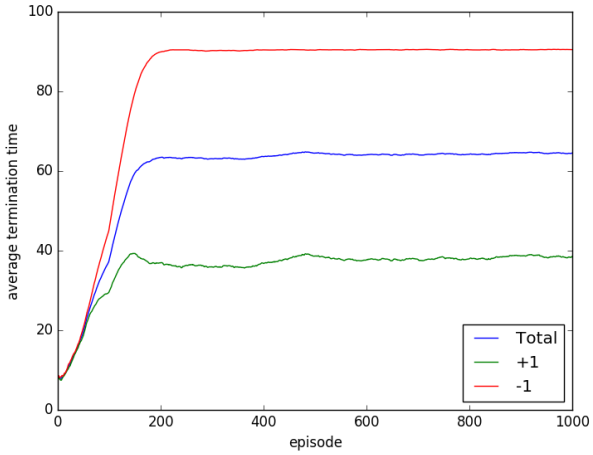

Figure 10: Unbiased prior, average episode length

We set a maximum episode length of 90 on these simulations. This means that after 90 steps, the agent will stop moving and end the period. We can see that when the goal reward is -1, the agent learns to stay in the system until reaching the maximal length and thus avoiding the negative reward.

5 Future Directions

We can modify our algorithm to incentivize the agent to learn faster. We can see in Figure 10 that even when the agent receives a positive reward for finding the goal, each episode lasts on average 40 time steps. We can modify the algorithm to give a slightly negative reward every time the agent lands in a state that is neither a hole or the goal and thus giving the agent an incentive to find the goal quickly.

In addition, when running LSVI_TD there may be clever ways of sampling the batches from the buffer that lead to better performance or more efficient computation. Possible methods include varying the batch size as the agent learns, sampling more from the episodes in which the agent receives a positive reward, or sampling from more recent episodes.